Concurrency-Enhancing Transformations for Asynchronous Behavioral Specifications: A Data-Driven Approach

John Hansen and Montek Singh
University of North Carolina
Chapel Hill, NC, USA
Most high-level async tools are syntax-directed (Haste/Balsa)

These tools are inadequate for designing high-speed circuits

Straightforward spec ➔ slow circuit

Fast circuits require significant effort

Need better tool support!

Most high-level async tools are syntax-directed (Haste/Balsa)

These tools are inadequate for designing high-speed circuits

Straightforward spec ➔ slow circuit

Fast circuits require significant effort

Need better tool support!
Our Contribution

“Source-to-Source Compiler”
- Rewrites specs to enhance concurrency
- Fully-automated and integrated into Haste flow
- Arsenal of several powerful optimizations:
  - parallelization, pipelining, arithmetic opt., communication opt.

Benefits:
- Up to 59x speedup (throughput) of implementation ... 
- ... 290x speedup with arithmetic optimization
- Or: Reduces design effort by up to 95% (lines of code)
  - with our method: high performance with low design effort
  - without our method: high performance requires significant effort!
Our Contribution

Our tool integrated as “preprocessor” to Haste compiler
- leverages Haste compilation and backend

Behavioral Spec → Compiler → Handshake Graph → TechMap → Netlist

Parallelize
Pipeline
Communication Opt.

Original Haste Flow
Our Contribution

4 concurrency-enhancing optimizations:

- **Parallelization**
  - remove unnecessary sequencing

- **Pipelining**
  - allow overlapped execution

- **Arithmetic Optimization**
  - decompose/restructure long-latency operations

- **Channel Communication Optimization**
  - re-ordering for increased concurrency

```plaintext
X?<<a,b,c,d>>;
```

```plaintext
e:=a+b; f:=c+d;
g:=f+1; h:=g*2;
k:=(e*f)*(g*h);
```

```plaintext
Y!k;
Z!e;
```
Our Contribution

Benefits of automatic code rewriting:

- Eases burden on designer
  - allows focus on functionality instead of perf.
  - greater readability → less chance of bugs

- Step towards design space exploration
  - selectively apply optimizations where needed...
  - ... based on a cost function (speed/energy/area)

- Backwards compatible with legacy code
  - simply recompile for high-speed implementation

---

**Designer's Code**

```plaintext
forever do
  IN?<<a,b,c,d,e,f>>;
  g := a * b;
  h := c * d;
  i := e * f;
  j := g + h;
  k := i * j;
  OUT!k
od
```

**Transformed Code**

```plaintext
&contextproc1 = proc (IN? chan ... & OUT! chan ...).
begin
  context : var <<...>>
  forever do
    IN?context;
    OUT!<<c, d, e, f, a * b>>
  od
end
```

```plaintext
&contextproc2 = ...
&contextproc3 = ...
...
&contextproc6 = ...
```
Solution Domain: Class of Specifications

- **Input Domain:** Requires “slack-elastic” specifications
  - Spec must be tolerant of additional slack on channels
  - Formally: deadlock-free, restriction on probes, ...
    [Manohar/Martin98]

- **Output:** Produces “data-driven” specifications
  - Pipelined: data drives computation, not control-dominated
  - Preserves top-level system topology, including cycles
  - Replaces each module with parallelized+pipelined version

- **Correctness model (slack elasticity):**
  - spec maintains original token order per channel
    - no guarantees about relative token order across channels
Solution Domain: Target Architectures

Can handle arbitrary topologies

Breaks down each module into smaller parts
Talk Outline

- Previous Work and Background
- Basic Approach
- Advanced Techniques
- Results
- Conclusion
Previous Work

“Spatial Computation” [Budiu 03]
- Convert ANSI C programs to dataflow hardware
- Spec language has inherent limitations
  - cannot model channel communication
  - no fork-join type of concurrency

Data-Driven Compilation [Taylor 08, Plana 05]
- New data-driven specification language
- “Push” instead of “pull” components
- Designer must still be skillful at writing highly concurrent specs
  - our approach effectively automates this by code rewriting
Previous Work

- **Peephole Optzn/Resynthesis** [Chelcea/Nowick 02, Plana 05]
  - improve concurrency at circuit and handshake levels
  - do not target higher-level (system-wide) concurrency

- **CHP Specifications** [Teifel 04, Wong 01]
  - translate CHP specs into pipelined implementations

- **Balsa/Haste \(\Leftrightarrow\) CDFG Conversion** [Nielsen 04, Jensen 07]
  - main goal is to leverage synchronous tools for resource sharing
  - some peephole optimizations only
**Background: Haste Language**

**Key language constructs:**

- Channel reads / writes
  - \( \text{IN}?x \quad / \quad \text{OUT}!y \)

- Assignments
  - \( a := \text{expr} \)

- Sequential / parallel composition
  - \( A ; B \quad / \quad A \parallel B \)

- Conditionals
  - if \( C \) then \( X \) else \( Y \) fi

- Loops
  - forever do
  - for
  - while

```plaintext
&fifo=proc(IN?chan byte &
          OUT!chan byte).

begin
  & x: var byte ff
  |    forever do
  |      IN?x;
  |      x:=x+1;
  |      OUT!x
  |    od
end
```
Background: Haste Compilation

A syntax-directed design flow for rapid development

Behavioral Spec

Compiler

Handshake Graph

TechMap

Netlist

&fifo=proc(IN?chan byte & OUT!chan byte).
  begin
    & x: var byte ff
    | forever do
      IN?x;
      OUT!x
    od
  end
Background: Haste Limitations

forever do
  IN?a;
  b:=f1(a);
  c:=f2(b);
  d:=f3(c);
  OUT!f4(d)
od

straightforward coding $\Rightarrow$ long critical cycles $\Rightarrow$ poor performance
**Basic Approach: Overview**

**Four step method:**

1. Input a behavioral specification
2. Perform parallelization on statements
3. Create a pipeline stage for each group of parallel statements
4. Produce new code incorporating these optimizations

```plaintext
proc(IN?chan byte & OUT!chan byte).
  forever do
    IN?a;
    1: b:=a*2;
    2: c:=b+5;
    3: d:=a+b;
    4: e:=c+d;
    5: f:=d*3;
    6: g:=f+e;
    OUT!g
  od
```

```
forever do (IN?a;
  OUT!<<a,a*2>>) od
...
forever do (IN?<<a,b>>;
  OUT!<<b+5,a+b>>) od
...
forever do (IN?<<a,b,c>>;
  OUT!<<c+d,d*3>>) od
...
forever do (IN?<<e,f>>;
  OUT!<<e+f>>) od
```
Parallelizing Transformation

Increases instruction-level concurrency

- Statements are re-ordered or parallelized

Original Example

```
proc(IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  b:=a*2;
  c:=b+5;
  d:=a+b;
  e:=c+d;
  f:=d*3;
  g:=f+6;
  OUT!g
od
```

Reduced Latency!

```
proc(IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  b:=a*2;
  c:=b+5;
  d:=a+b;
  e:=c+d;
  f:=d*3;
  g:=f+6;
  OUT!g
od
```
forever do
  IN?a;
  b:=a*2;
  (c:=b+5 ||
   d:=a+b);
  (e:=c+d ||
   f:=d*3);
  g:=f+e;
  OUT!g
od

Algorithm:
- Generate a dependence graph
- Perform a topological sort
  (group parallelizable statements)
- Sequence parallel groupings
Parallelizing: What About Cycles?

Cycles are collapsed into atomic nodes.

Parallelization is performed recursively.

```
forever do
  1: IN1?a;
  2: IN2?b;
  3: c:=c+1;
  4: d:=a+c;
  5: e:=b+c;
  6: c:=c+d+e;
  7: OUT!c
od
```

```
forever do
  1: (IN1?a ||
  2: IN2?b);
  3: (c:=c+1;
  4: (d:=a+c ||
  5: e:=b+c);
  6: c:=c+d+e);
  7: OUT!c
od
```
Pipelining Transformation

proc(IN?chan byte
     & OUT!chan byte).
forever do
  IN?a;
  1: b:=a*2;
  2: c:=b+5;
  3: d:=a+b;
  4: e:=c+d;
  5: f:=d*3;
  6: g:=f+e;
  OUT!g
od

Original Example

 Allows execution to overlap
 Control is distributed

Increased Throughput

Stage I (IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  OUT!<<a,a*2>>
od
...
Stage II (IN?chan byte & OUT!chan byte).
forever do
  IN<<a,b>>;
  OUT!<<a,b,b+5>>
od
...
Challenge: Modifying the flow of data

- How to communicate data?
  - data needs to flow through channels, not variables

- Which data to communicate?
  - transmit only necessary data (i.e., live values) to save area
  - we call this the context
Pipelining Transformation

Three step solution:

- **Compute IN-set:**
  - all values *produced* in or prior to a stage

- **Compute OUT-set:**
  - all values *consumed* in later stages

- **Compute context:**
  - all values *produced* in or prior to this stage that are *consumed* in later stages

\[
\begin{align*}
\text{IN}_1 &= \text{VAR}_1 \\
\text{IN}_x &= \text{IN}_{x-1} + \text{VAR}_x \\
\text{OUT}_N &= \emptyset \\
\text{OUT}_x &= \text{OUT}_{x+1} + \text{VAR}_{x+1} \\
\text{context}_x &= \text{IN}_x \cap \text{OUT}_{x-1}
\end{align*}
\]
Pipelining: Connecting the Stages

Each stage is connected by communicating values across channels using channel actions.

Connect a stage $x$ with its successor:
- communicate the values contained in context $x+1$. 

Diagram: Six stages with input 'a' and output 'g'.
Pipelining: Source to Source

forever do
    IN?a;
    b:=a*2;
    c:=b+5;
    d:=a+b;
    e:=c+d;
    f:=d*3;
    g:=f+e;
    OUT!g
od

forever do (IN?a;
    OUT!<<a,a*2>>) od

... forever do (IN?<<a,b>>;
    OUT!<<a,b,b+5>>) od

... forever do (IN?<<a,b,c>>;
    OUT!<<c,a+b>>) od

... forever do (IN?<<c,d>>;
    OUT!<<d,c+d>>) od

... forever do (IN?<<d,e>>;
    OUT!<<e,d*3>>) od

... forever do (IN?<<e,f>>;
    OUT!<<f+e>>) od
Pipelining: Reducing Control Overheads

Single Large Cycle

Several Smaller Cycles -> Higher Throughput
proc(IN?chan byte & OUT!chan byte).
forever do
    IN?a;
    b:=a*2;
    c:=b+5;
    d:=a+b;
    e:=c+d;
    f:=d*3;
    g:=f+e;
    OUT!g
od

Gain benefits of both optimizations

Stage 1(IN?chan byte & OUT!chan byte).
forever do
    IN?a;
    OUT!<<a,a*2>>
od
...
Stage 2(IN?chan byte & OUT!chan byte).
forever do
    IN<<a,b>>;
    OUT!<<b+5,a+b>>
od
...
Talk Outline

- Introduction
- Background
- Basic Approach
- Advanced Techniques
  - Arithmetic Optimization
  - Handling Conditionals and Loops
  - Communication Optimization
- Results
- Conclusion
Arithmetic Optimization

- Perform parallelization and pipelining at a sub-statement level

- 3 specific optimizations:
  - Balancing Expression Trees
  - Expression Pipelining
  - Operator Pipelining
Balancing Expression Trees

- Restructures expressions into balanced trees
  - Essentially: parallelize at level of sub-expressions

Example:

- **Original**
  - \(q:=a+b+c+d\)
  - 3 sequential sums

- **Balanced**
  - \(q:= (a+b)+(c+d)\)
  - 2 parallel sums in sequence with third

Reduced Latency
Expression Pipelining

- Decompose complex expressions into simpler ones
  - Essentially: pipelining at the *expression* level

**Example:**

- **Original**
  - \( q := a \times b \times c - d \)

- **Decomposed**
  - \( q_1 := a \times b \)
  - \( q_2 := q_1 \times c \)
  - \( q := q_2 - d \)

**Reduced Cycle Time ➔ Higher Throughput**
Operator Pipelining

- Decompose a long-latency arithmetic operation into smaller pieces
  - Essentially: pipelining at the operator level

\[ a := b + c \]

- \[ a_1 := b_1 + c_1 \]
- \[ a_2 := b_2 + c_2 \]
- \[ a_3 := b_3 + c_3 \]
- \[ a_4 := b_4 + c_4 \]
- \[ a_2 += \text{carry} \]
- \[ a_3 += \text{carry} \]
- \[ a_4 += \text{carry} \]
Conditional Constructs

Several options for handling conditionals

- Conditional Assignment
- Late Decision (speculation)
- Early Decision

```
y := if a > b then a else b;
x := if a > b then y - b else y - a
```

- Conditional Assignment
  
  ```
y := if a > b then a else b;
x := if a > b then y - b else y - a
```
Handling Loops

**Challenge: Significant performance bottleneck**
- Circuit-level pipelining cannot speed up *single-token loops*
- Each loop acts as a single unpipelined high-latency stage

**Our approach**
- Use parallelization + arithmetic optimization to lower loop latency
  - decrease in latency = increase in overall throughput
- Use loop unrolling to further help with parallelization
- Transform into “multi-token” loops
  - Plan to incorporate in future [Gill, Hansen, Singh 06]
Communication Optimization

Challenge: Channel actions complicate optimizations

- Unlike other statements, channel actions tricky to reorder
  - Besides dependencies within module...
  - Dependencies and synchronization with other modules

Solution:

- Conservative approach: strictly maintain order of channel actions
- Our proposed approach: safely re-order channel actions
  - Introduced a constraint to guarantee safety
  - Benefit: can lead to higher concurrency
Example: Benefit of reordering channel actions

M: (Original)
forever do
    A?a;
    B?b;
    C?c;
    disc:=b*b-4*a*c;
    X!disc;
    Y!(2*a)
od

M: (Optimized)
forever do
    (A?a||B?b||C?c);
    (Y!(2*a)||
    disc:=b*b-4*a*c);
    X!disc
od

Outputs are produced earlier!
Communication Optimization

Challenge: arbitrary re-orderings can introduce deadlock!

Original order: Channel actions succeed
Communication Optimization

Challenge: arbitrary re-orderings can introduce deadlock!

After reordering: Deadlock is introduced!
Systematic approach for determining legal re-orderings:

- Build a directed graph
  1. Make a node for each channel
  2. Add edges for data dependence
  3. Add edges for sequencing

- New sequencing is legal if graph does not contain a cycle
  - cycle = deadlock
Talk Outline

- Introduction
- Background
- Basic Approach
- Advanced Techniques
- Results
- Conclusion
Experimentation Setup

- **Our approach implemented in Java**
  - integrated as pre-processor into Haste flow

- **Simulation performed using Haste design flow**
  - 8 non-trivial examples
    - includes straight-line code, conditionals, and loops

- **Evaluated:**
  - throughput
  - latency
  - area
  - design effort
Throughput improvement

Parallelization

- 2.0x add
- 1.9x comm
- 2.1x fir
- 2.0x ode
- 1.1x root
- 1.0x quad
- 1.0x tea
- 1.0x tea2

Arithmetic Pipelining

- 5.5x add
- 32 com
- 16 fir
- 8 ode
- 4 root
- 4 quad
- 4 tea
- 4 tea2

- 59.2x overall (parallelization + pipelining)
- 23.3x through parallelization
- 2.2x through pipelining
- 5.2x through arithmetic pipelining (overall: 290x)
Latency, Circuit Area, and Effort

Latency: Pipelining

Area Overhead

Reduction in Designer Effort

- Latency generally reduced by parallelization, and increased by pipelining
- Area increases with depth of pipelining
- Design effort ~20-95% lower
Conclusion

Developed a source-to-source compilation approach:

- Powerful set of optimizations
  - parallelization & pipelining
  - arithmetic & communication optimization
- Throughput speedup of up to 59x
  - ... up to 290x with arithmetic optimization
  - Or: 95% design effort reduction
- Integrated into Haste design flow

Future Work:

- full dataflow implementation
- explore slack matching issues
- loop pipelining
- large example (simple processor)